

dplan: agende condivise client-server

Progetto finale (terzo frammento) del corso di LCS 2007/08

Indice

1	Introduzione	1
1.1	Materiale in linea	2
1.2	Struttura del progetto e tempi di consegna	2
1.3	Valutazione del progetto	2
2	Il progetto: dplan	3
3	Il server	4
4	Il client	4
5	Protocollo di interazione client-server	6
5.1	Formato dei messaggi	6
5.2	Messaggi da Client a Server	7
5.3	Messaggi da Server a Client	7
6	Istruzioni	8
6.1	Materiale fornito dai docenti	8
6.2	Cosa devono fare gli studenti	8
7	Parti Opzionali	8
8	Codice e documentazione	8
8.1	Vincoli sul codice	8
8.2	Formato del codice	9
8.3	Relazione	10

1 Introduzione

Il corso di Laboratorio di Programmazione Concorrente e di Sistema (AA538 – 6 crediti) prevede lo svolgimento di due suite di esercizi intermedi (frammento1 e frammento 2) e di un progetto finale individuale. Il progetto finale è descritto in questo documento.

Il progetto consiste nello sviluppo del software relativo a **dplan** (Distributed PLANning): un sistema client-server che realizza un pool di agende condivise

fra diversi utenti. Il software viene sviluppato e documentato utilizzando gli strumenti, le tecniche e le convenzioni presentati durante il corso.

1.1 Materiale in linea

Tutto il materiale relativo al corso può essere reperito sul sito Web:

<http://www.cli.di.unipi.it/doku/doku.php/lcs/lcs08/start>

Il sito verrà progressivamente aggiornato con le informazioni riguardanti il progetto (es. FAQ, suggerimenti, avvisi), sul ricevimento e simili. Il sito è un Wiki e gli studenti possono registrarsi per ricevere automaticamente gli aggiornamenti delle pagine che interessano maggiormente. In particolare consigliamo a tutti di registrarsi alla pagina delle 'FAQ' e degli 'avvisi urgenti'.

Eventuali chiarimenti possono essere richiesti consultando i docenti di LCS durante l'orario di ricevimento, le ore in laboratorio e/o per posta elettronica.

1.2 Struttura del progetto e tempi di consegna

Il progetto deve essere sviluppato da un singolo studente individualmente e può essere consegnato entro il 1 Febbraio 2009.

La consegna del progetto avviene *esclusivamente* attraverso il target **consegna** del **Makefile** contenuto nel kit di sviluppo del progetto. Eventualmente, una copia del tar creato dal target **consegna** può essere allegata ad un normale messaggio di posta elettronica. Le consegne sono seguite da un messaggio di conferma da parte del docente all'indirizzo di mail da cui la consegna è stata effettuata. Se non ricevete tale messaggio entro qualche giorno lavorativo contattate il docente.

*I progetti che non rispettano il formato o non consegnati con il target **consegna** non verranno accettati.*

La data ultima di consegna dei due frammenti e del progetto finale è il 01/02/09. Dopo questa data gli studenti dovranno svolgere i tre frammenti di progetto previsti per il corso 2008/09.

Inoltre, gli studenti che consegnano una versione sufficiente del progetto finale entro il 31 luglio 2008 accumuleranno il terzo bonus di 2, che contribuisce al voto finale con le modalità illustrate nelle slide introduttive del corso (vedi sito).

1.3 Valutazione del progetto

Al progetto viene assegnato un punteggio da 0 a 26 di cui 6 punti esclusivamente determinati dalla *qualità della documentazione allegata*. La valutazione del progetto è effettuata in base ai seguenti criteri:

- motivazioni, originalità ed economicità delle scelte progettuali
- strutturazione del codice (suddivisione in moduli, uso di makefile e librerie etc.)
- efficienza e robustezza del software
- modalità di testing

- aderenza alle specifiche
- qualità del codice C e dei commenti
- chiarezza ed adeguatezza della relazione (vedi Sez. 8.3)

La prova orale tenderà a stabilire se lo studente è realmente l'autore del progetto consegnato e verterà su tutto il programma del corso. Il voto dell'orale (ancora da 0 a 26) fa media con la valutazione del progetto per delineare il voto finale. In particolare, l'orale comprenderà

- una discussione delle scelte implementative del progetto e dei frammenti
- l'impostazione e la scrittura di script bash e makefile
- l'impostazione e la scrittura di programmi C + PosiX non banali (sia sequenziali che concorrenti)
- domande su tutto il programma presentato durante il corso.

Casi particolari Gli studenti lavoratori iscritti alla laurea triennale in Informatica possono consegnare i due frammenti e il progetto finale in un'unica soluzione in qualsiasi momento dell'anno ed essere valutati con votazione da 0 a 30. In questo caso è necessaria la certificazione da consegnare al docente.

Gli studenti che svolgono il progetto per abbreviazioni delle nuove lauree specialistiche sono invitati a contattare il docente.

Gli studenti iscritti ai vecchi ordinamenti (nei quali il voto di Lab. 4 contribuiva al voto di SO) avranno assegnato un voto in trentesimi che verrà combinato con il voto del corso di SO corrispondente secondo modalità da richiedere ai due docenti coinvolti.

2 Il progetto: dplan

Lo scopo del progetto è lo sviluppo di un sistema client server che realizza un pool di agende condivise fra un insieme di utenti. Il sistema è formato da due programmi C: un server (**dserver**) ed un client (**dplan**). Il server gestisce un insieme di agende. Ogni agenda ha un nome univoco di al più 20 caratteri.

Ogni agenda è memorizzata in un file di testo con lo stesso nome. Nel file agenda sono memorizzati gli eventi secondo il formato

```
gg-mm-aaaa utente#descrizione
```

in cui il primo campo rappresenta la data dell'evento, **utente** è una stringa di al più 8 caratteri che identifica l'utente che ha registrato l'evento e **descrizione** costituisce una descrizione sintetica (fino a fine riga). La lunghezza massima di una descrizione è 80 caratteri.

I client possono richiedere al server di creare una nuova agenda, aggiungere o cancellare eventi d agende esistenti e rimuovere agende vuote. Inoltre possono richiedere informazioni sugli eventi presenti in una data agenda in un certo giorno o mese.

3 Il server

Il server viene attivato da shell con il comando

```
$ dserver diragende
```

dove **diragende** è il path della directory che contiene i file agenda. Il server lavora in foreground.

Se **diragende** esiste, la apre in lettura e scrittura e per ogni file agenda **aname** contenuto nella directory:

- controlla che il formato sia compatibile con quello descritto nella sezione precedente,
- se il controllo ha successo carica il contenuto dell'agenda **aname** all'interno di una lista in memoria centrale

Se **diragende** non esiste viene creato. All'attivazione, il server crea anche una socket AF_UNIX nella directory **./tmp**

./tmp/dsock

su cui i client apriranno le connessioni con il server.

Il server può essere terminato *gentilmente* inviando un segnale di **SIGTERM**. All'arrivo di questo segnale il server deve eliminare la socket **dsock** ed eventuali file temporanei, terminare i thread che lo compongono in modo che eventuali richieste pendenti da parte dei client vengano completate correttamente ed uscire. È necessario anche che tutte le agende presenti in memoria vengano scritte nel corrispondente file in **diragende**.

Quando il server è attivo accetta dai client richieste di

- creazione di una nuova agenda
- aggiunta/rimozione di eventi da una agenda
- richiesta degli eventi in un certo giorno o mese.
- rimozione dell'agenda (solo se vuota)

Il protocollo di interazione è descritto nella Sezione 5.

4 Il client

Il client è un comando Unix che può essere invocato con diverse modalità ed opzioni

```
dplan -c aname
```

```
-> crea una nuova agenda di nome ''aname''
```

```
dplan -q aname
```

```
-> rimuove una agenda di nome ''aname'' (solo se vuota)
```

```
dplan agenda -d gg-mm-aaaa -u utente#descrizione
```

```
-> inserisce in ''agenda'' un nuovo evento
```

-> -d specifica la data
-> -u specifica l'utente che sta effettuando la registrazione
e la descrizione dell'evento

dplan agenda -g gg-mm-aaaa
-> richiede gli eventi registrati per un certo giorno
su 'agenda' e li stampa sullo stdout

dplan agenda -m mm-aaaa
-> richiede gli eventi registrati su 'agenda' per un
certo mese e li stampa sullo stdout

dplan agenda -r pattern
-> elimina tutti gli eventi di 'agenda' che contengono
'pattern' come sottostringa in un qualsiasi campo

dplan
-> stampa un messaggio di uso

Appena attivato, il client effettua il parsing delle opzioni, e se il parsing è corretto cerca di collegarsi con il server per un massimo di 5 tentativi a distanza di 1 secondo l'uno dell'altro. Se il collegamento ha successo interagisce con il server per portare a termine l'operazione richiesta.

Segue una esemplificazione delle varie operazioni. Creazione/distruzione agenda

```
$$ ./dplan -c ciccio
dplan: ciccio: Created
$$ ./dplan -c ciccio
dplan: ciccio: Cannot create, agenda already present
$$ ./dplan -q ciccio
dplan: ciccio: Removed
$$ ./dplan -q test1.dat
dplan: test1.dat: Agenda not empty, cannot remove
```

Ecco, invece, un esempio di richiesta giornaliera:

```
$$ ./dplan test1.dat -g 01-08-2008
dplan: test1.dat: 01-08-2008 cinzia#Pagare rata macchina
$$
```

o mensile

```
$$ ./dplan test1.dat -m 06-2008
dplan: test1.dat: 01-06-2008 cinzia#Pagare rata macchina
dplan: test1.dat: 04-06-2008 davide#Scrivere a Nadia
dplan: test1.dat: 04-06-2008 davide#Telefonare zia luigina
$$
```

Si noti che gli eventi vengono forniti in ordine e che per la stessa data ed utente si effettua un ordinamento lessicografico sulle descrizioni.

Ecco la rimozione degli eventi che corrispondono ad un certo pattern:

```
$$ ./dplan test1.dat -r luigi
dplan: luigi: Success
$$ ./dplan -r luigi priscilla
dplan: priscilla: Agenda not existent
```

nel primo caso la rimozione è avvenuta nel secondo la richiesta era errata perchè riguardava un'agenda non esistente.

Infine l'introduzione di un nuovo evento avviene con

```
$$ ./dplan test1.dat -d 02-12-2008 -u "maria#Parrucchiere (ore 14:00)"
dplan: 02-12-2008: Success
```

5 Protocollo di interazione client-server

Server e client interagiscono utilizzando *socket AF_INET*.

I client si connettono al server attraverso il socket `./tmp/dplan`¹. La socket viene creata all'avvio del server.

5.1 Formato dei messaggi

I messaggi scambiati fra server e client hanno la seguente struttura:

```
typedef struct {
    char type;
    unsigned int length;
    char* buffer;
} message_t;
```

Il campo `type` è un `char` (8 bit) che contiene il tipo del messaggio spedito. `type` può assumere i seguenti valori:

```
#define MSG_ERROR      'E'
#define MSG_OK         'O'
#define MSG_MKAGENDA   'C'
#define MSG_RMAGENDA   'Q'
#define MSG_INSERT     'D'
#define MSG_RMPATTERN  'R'
#define MSG_EMESE      'M'
#define MSG_EGIORNO    'G'
```

Il campo `length` è un `unsigned int` (32 bit) che indica la dimensione del campo `buffer`. Se `buffer` è una stringa il suo valore comprende anche il terminatore di stringa `'\0'` finale che deve essere presente nel `buffer`. Il campo `length` Vale 0 nel caso in cui il campo `buffer` non sia significativo. Il campo `buffer` è un puntatore a un buffer di caratteri.

¹Sarebbe più logico creare la socket nella directory `/tmp/` di sistema ma per non avere problemi di interazioni indesiderate fra progetti diversi sulle macchine del cli è meglio creare tutte le pipe in una directory `./tmp/` locale alla directory di progetto

5.2 Messaggi da Client a Server

Nei messaggi spediti dal Client al Server, il campo **type** può assumere i seguenti valori:

MSG_MKAGENDA in questo caso il buffer contiene il nome dell'agenda. Il server risponde con **MSG_OK** se tutto è andato bene o **MSG_ERROR** e l'eventuale messaggio di errore se c'è stato un errore.

MSG_RMAGENDA in questo caso il buffer contiene il nome dell'agenda. Il server risponde con **MSG_OK** se tutto è andato bene o **MSG_ERROR** e l'eventuale messaggio di errore se c'è stato un errore.

MSG_MKINSERT in questo caso il buffer contiene nome agenda, data, nome utente e descrizione, tutti separati da un carattere \0

agenda\0data\0utente\0descrizione\0

tutto terminato da \0. Il server risponde con **MSG_OK** se tutto è andato bene o **MSG_ERROR** e l'eventuale messaggio di errore se c'è stato un errore.

MSG_RMPATTERN in questo caso il buffer contiene nome agenda, e pattern separati da \0

agenda\0pattern\0

tutto terminato da \0. Il server risponde con **MSG_OK** se tutto è andato bene o **MSG_ERROR** e l'eventuale messaggio di errore se c'è stato un errore.

MSG_EGIORNO in questo caso il buffer contiene nome agenda, e data separati da \0

agenda\0data\0

tutto terminato da \0. Il server risponde con una serie di **MSG_OK** (ognuno contenente un evento da visualizzare) se tutto è andato bene o **MSG_ERROR** e l'eventuale messaggio di errore se c'è stato un errore.

MSG_EMесе in questo caso il buffer contiene nome agenda, e mese separati da \0

agenda\0mese\0

tutto terminato da \0. Il server risponde con una serie di **MSG_OK** (ognuno contenente un evento da visualizzare) se tutto è andato bene o **MSG_ERROR** e l'eventuale messaggio di errore se c'è stato un errore.

5.3 Messaggi da Server a Client

Nei messaggi spediti dal Server a ciascun Client, il campo **type** può assumere i seguenti valori:

MSG_ERROR Messaggio di errore. Questo tipo di messaggio viene spedito quando si riscontra un errore. Il campo **buffer** contiene la stringa che definisce l'errore riscontrato.

MSG_OK Segnala che la richiesta è andata a buon fine. Il formato dipende dalla richiesta.

6 Istruzioni

6.1 Materiale fornito dai docenti

Nel kit del progetto vengono forniti

- funzioni di test e verifica per libldplan
- `makefile` per test e consegna
- file di intestazione (.h) con definizione dei prototipi e delle strutture dati
- vari README di istruzioni

6.2 Cosa devono fare gli studenti

Gli studenti devono:

- leggere *attentamente* i README e capire il funzionamento del codice fornito dai docenti
- implementare le funzioni richieste e testarle
- verificare le funzioni con i test forniti dai docenti (attenzione: questi test vanno eseguiti su codice già corretto e funzionante altrimenti possono dare risultati fuorvianti o di difficile interpretazione)
- preparare la *documentazione*: vedi la Sez. 8.
- sottomettere il progetto esclusivamente utilizzando il makefile fornito e seguendo le istruzioni nel README.

7 Parti Opzionali

Opzionalmente possono essere realizzate funzionalità ed opzioni in più rispetto a quelle richieste.

Per queste opzioni deve essere decisa un'opportuna sintassi da riportare nella relazione.

8 Codice e documentazione

In questa sezione, vengono riportati alcuni requisiti del codice sviluppato e della relativa documentazione.

8.1 Vincoli sul codice

Makefile e codice devono compilare ed eseguire CORRETTAMENTE su (un sottinsieme non vuoto del) le macchine del CLI. Il README (o la relazione) deve specificare su quali macchine è possibile far girare correttamente il codice. Inoltre, se si usano software e librerie non presenti al CLI: (1) devono essere presenti nel tar TUTTI i file necessari per l'installazione in locale del/i tool e

(2) devono essere presenti nel *makefile* degli opportuni target per effettuare automaticamente l'installazione in locale. Se questa condizione non è verificata il progetto non viene accettato per la correzione.

La stesura del codice deve osservare i seguenti vincoli:

- la compilazione del codice deve avvenire definendo delle regole appropriate nella parte iniziale del *makefile* contenuto nel kit;
- il codice deve compilare senza errori o warning utilizzando le opzioni `-Wall -pedantic`
- se non siamo sicuri della presenza del carattere terminatore di stringa NON devono essere utilizzate funzioni per la manipolazione delle stringhe che *non* limitano il numero di caratteri scritti/manipolati, ad esempio la `strcpy()` deve essere evitata a favore della `strncpy()` in cui è possibile fissare il massimo numero di caratteri copiati (per le motivazioni consultare il man in linea)
- NON devono essere utilizzate funzioni di temporizzazioni quali le `sleep()` o le `alarm()` per risolvere problemi di race condition o deadlock fra i processi. Le soluzioni implementate devono necessariamente funzionare qualsiasi sia lo scheduling dei processi coinvolti
- DEVONO essere usati dei nomi significativi per le costanti nel codice (con opportune `#define` o `enum`)

8.2 Formato del codice

Il codice sorgente deve adottare una convenzione di indentazione e commenti chiara e coerente. In particolare deve contenere

- una intestazione per ogni file che contiene: il nome ed il cognome dell'autore, la matricola, il nome del programma; dichiarazione che il programma è, in ogni sua parte, opera originale dell'autore; firma dell'autore.
- un commento all'inizio di ogni funzione che specifichi l'uso della funzione (in modo sintetico), l'algoritmo utilizzato (se significativo), il significato delle variabili passate come parametri, eventuali variabili globali utilizzate, effetti collaterali sui parametri passati per puntatore etc.
- un breve commento che spieghi il significato delle strutture dati e delle variabili globali (se esistono);
- un breve commento per i punti critici o che potrebbero risultare poco chiari alla lettura
- un breve commento all'atto della dichiarazione delle variabili locali, che spieghi l'uso che si intende farne

8.3 Relazione

La documentazione del progetto consiste nei commenti al codice e in una breve relazione (massimo 10 pagine) il cui scopo è quello di descrivere la struttura complessiva del lavoro svolto. La relazione *deve rendere comprensibile il lavoro svolto ad un estraneo, senza bisogno di leggere il codice se non per chiarire dettagli implementativi*. In pratica la relazione deve contenere:

- le principali scelte di progetto (strutture dati principali, algoritmi fondamentali e loro motivazioni)
- la strutturazione del codice (logica della divisione su più file, librerie etc.)
- la struttura del server e del client
- la struttura dei programmi di test
- le difficoltà incontrate e le soluzioni adottate
- quanto altro si ritiene essenziale alla comprensione del lavoro svolto
- README di istruzioni su come compilare/eseguire ed utilizzare il

La relazione deve essere in formato PDF.